ANNUAL JOURNAL OF ELECTRONICS, 2013, ISSN 1314-0078

Task Parallel Paradigms: a Comparative Case Study

Piet Cordemans, Jeroen Boydens and Eric Steegmans

Abstract – **Various tasks can run efficiently in parallel on current processor architectures. However, writing software to coordinate workflow between tasks is a challenge. In this paper, three task parallel paradigms are evaluated. Two are iterator-based, namely lightweight Tasks, which encompass little overhead, and Futures providing support for continuations. The third paradigm, Reactive programming is based on observers. In a case study, based on processing data streams in an embedded system, we evaluate these paradigms on efficiency, expressiveness and composability.**

Keywords – **embedded systems, task parallelism, stream processing**

I. Introduction

More and more multicore processing is adapted in computational-intensive embedded systems. In this respect the embedded software paradigms need to shift to optimally exploit the inherent parallelism. In this paper we discuss task parallelism, more specifically three task parallel paradigms. In Section II, three criteria are defined to evaluate those paradigms. Section III describes a case study, based on stream processing. Consequently an evaluation is given in Section IV, according to metrics related to the defined criteria. Finally, related and future work are discussed.

A. Task parallelism

Task parallelism is defined as executing different tasks in parallel across multiple processor cores. Basically, a thread is forked and two execution paths become available. Based on the process ID, two independent tasks can be executed. In contrast, data parallelism is a form of parallelism in which multiple processors cores execute the same task in parallel each on a separate part of the data set. These two approaches can be combined whereas tasks executed in parallel can be performed in parallel with other tasks.

B. Task-based paradigms

Many paradigms exist in order to implement task parallelism. We selected the paradigms, which have already been adapted in parallel computing and applied them in an embedded context.

First are Tasks as a unit of concurrent computation. A Task is a lightweight concurrent thread, which has its own scheduler [5]. This scheduler allows to optimally employ the underlying processor architecture. Fundamentally, this scheduler manages a pool of threads, whose number typically corresponds to the number of available cores. The scheduler strives to distribute work evenly across the threads in the thread pool. When a thread has no Tasks in its local task queue it tries to steal a Task from the queue of another thread, hence the name work-stealing.

Whereas Tasks provide an execution mechanism, the paradigm does not offer mechanisms for notifications or synchronization constructs. In order to execute safely in a shared memory context, Tasks need to employ basic synchronization constructs, such as locks or barriers. In this respect, the implementation of a continuation control mechanism requires a significant amount of boilerplate code.

The Future paradigm is the second discussed in this paper. A Future is a data structure, which facilitates the definition of continuations [8]. It is an object, which acts as a proxy for a result that has yet to be computed. The main opportunity a Future offers, is non-blocking calls. While the value can be asynchronously computed, the caller can continue until it needs to resolve the effective value. At that point, the value the Future represents either has already been computed, thus immediately returning without blocking control flow. In the other case, the value was not computed yet and the caller blocks until the value becomes available.

The main advantage of Futures is that they can be passed around or can be archived, without the need of calculating the result. However retrieving the result before it has been calculated induces overhead, as a callback to the continuation must be registered.

The final paradigm, Reactive programming, takes a different approach to control flow. Rather than executing a series of statements, the behavior is declaratively described [11]. The underlying execution mechanism will automatically evaluate changes in data flow, called events. In this perspective, an analogy can be made with the Observer pattern [4]. The events are published as a series of notifications, whereas an observer can subscribe to them. In this observer the behavior is defined. Following this analogy, the Task and Future-based paradigms contrast with Reactive programming, as the first follow the Iterator pattern and the last is related to the Observer pattern.

II. Criteria

In this study, we consider three criteria to evaluate the task parallel paradigms. On the one hand, with respect to the static properties of code, (1) expressiveness and (2) composability are considered. On the other hand, (3) efficiency of the runtime behavior is evaluated.

A. Expressiveness

P. Cordemans, and J. Boydens are with the Department of Industrial Sciences and Technology, KHBO.

P. Cordemans, J. Boydens, and E. Steegmans are with the Department of Computer Science, KU Leuven, iMinds – DistriNet research group, e-mail: {Piet.Cordemans, Jeroen.Boydens, Eric.Steegmans}@cs.kuleuven.be

Expressiveness is a criterion based on simplicity of defining a concept in code [3]. Task parallel concepts of importance are notifications and continuations. The need of boilerplate code, or lack thereof, defines whether the implementation of the paradigm facilitates the use these task parallel concepts.

B. Composability

With respect to code reuse, composability is a key criterion. It is defined by the dependencies of code on other software. When the number of dependencies is high, the code is said to be strongly coupled. Such code is not only hard to maintain, but also more difficult to test.

C. Efficiency

High level language concepts typically have an effect on the efficiency of code. On the one hand, this can be positive, for instance when a compiler heavily optimizes generated code. On the other hand the language concept can have a negative effect, whereas levels of indirection burden the performance. In the latter case, the effect on the runtime has to be considered relatively to the gain in expressiveness or composability.

III. CASE STUDY

In order to rule out unrelated variables in the comparative case study, a common platform was chosen, on which each paradigm was represented. The .NET environment was selected, as it provides the necessary libraries and language features to support each paradigm.

First, the task paradigm was implemented using the Task Parallel Library (TPL) [5], which allows to create Tasks, synchronize between tasks and manage the underlying threads. Next, the Future paradigm was implemented as a language feature in C# version 5 called async / await [8]. Async identifies the method in which the continuation is defined. While the await keyword provides synchronization, in case the Future has not been completed. Finally, the reactive paradigm is provided in the Reactive Extensions (Rx) library [13], which is based on an interface called IObserver. This library provides observable collections and LINQ-style query operators to process data in these collections.

The case study was conducted on an ultrasound scanning system for contactless material quality control. By measuring the amplitude of ultrasonic waves through the material under test, the structure of the material can be determined. A module consists of four sensors and a measurement setup consists of a number of modules depending on the size of the material under test. Each measurement contains the maximum value of the cross-correlation between the sent and received signal and the position of that point in the signal. Metadata is added to uniquely identify each measurement, by index, channel and module number. This setup generates multiple streams of data, which are sent over Ethernet to a computer for processing.

A. Streams

Regardless of the paradigm, the application is divided in two tiers, namely a communication tier and a processing tier, as shown in Figure 1.

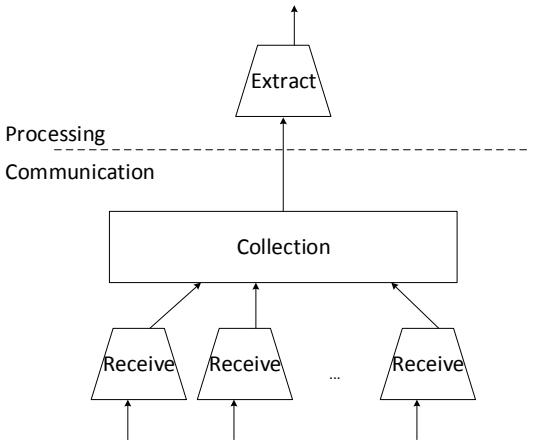

FIGURE 1. APPLICATION ARCHITECTURE

Each Ethernet interface generates a stream of multiple connections. As it is not guaranteed that a packet is completely delivered during a single connection, it needs to be buffered. Buffering aggregates the Ethernet streams in a single bytestream. As packet size is variable and defined in the header of the packet, the parser needs to determine packet size first, before proceeding. This leads to a different implementation of the application between the paradigms.

Both Tasks and Future based applications used an iterator-based buffer implementation to store received data while the packets are parsed. This allowed us to eagerly evaluate the header of the packet, thus determining packet size and continuing with data point extraction. On the other hand the reactive implementation of an observable buffer required to close the buffer before data was processed. However, in order to determine when the buffer should be closed the data header needs to be processed to extract packet size. In order to deal with this problem, reactively parsing the bytestream was implemented in two stages. In the first stage a binary parser extracted the number of measurements, subsequently it parsed the result to create the effective set of data.

B. Methods

To measure expressiveness, composability and efficiency as determined in Section II, Lines Of Code (LOC), cyclomatic complexity, class-coupling and time to process a given set of data are determined.

Complexity of the code is expressed in two measures, namely LOC and cyclomatic complexity. On the one hand, a lower number of logical lines of code, indicates that the code is more comprehensible. On the other hand, cyclomatic complexity is a metric developed by McCabe [6], which is proportional to the size of the control flow graph. The complexity of the control flow graph (M) is defined by the number of edges (E), number of nodes (N), and number of connected components (P), as in Eq. 1:

$$M = E - N + 2P \qquad (1)$$

Composability is measured through determining class-coupling. This is a number between 0.66 indicating a loosely coupled class, and 1.0 corresponding to a strongly coupled class. Given the number of input parameters (i), the number of output parameters (o), the number of modules called (w), the number of modules calling the method (r), class coupling (C) is determined as in Eq. 2:

$$C = 1 - \frac{1}{i + o + w + r} \quad (2)$$

In contrast to expressiveness and composability, efficiency is dependent on the run-time environment, namely the machine, operating system, framework and implementation of the paradigm.

C. Measurements

In Figures 2, 3 and 4 the methods are listed which were considered in the measurements. There is a distinction between the communication methods, i.e. Connect, Disconnect, Read and Stop and the processing method, i.e. Extract. The purpose of these two groups of methods is distinct, as communication is related to a stream of events, whereas processing has a significant body of work.

In Figure 2 the executable logical Source Lines Of Code (SLOC –L) are given for the respective methods with each of the paradigms.

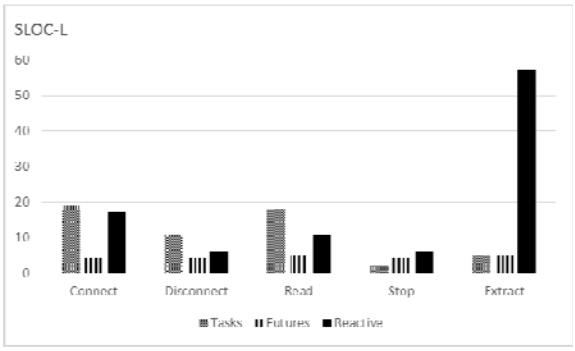

FIGURE 2. LINES OF CODE

Figure 3 indicates Cyclomatic complexity of each of the methods in respect to their implementation of the paradigms.

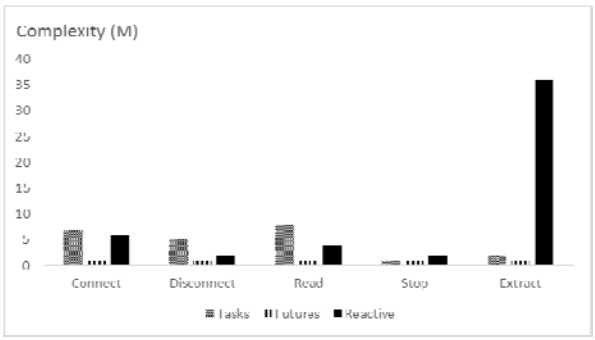

FIGURE 3. CYCLOMATIC COMPLEXITY

The final static metric, class-coupling for each of the methods as implemented according to the respective paradigms, is given in Figure 4.

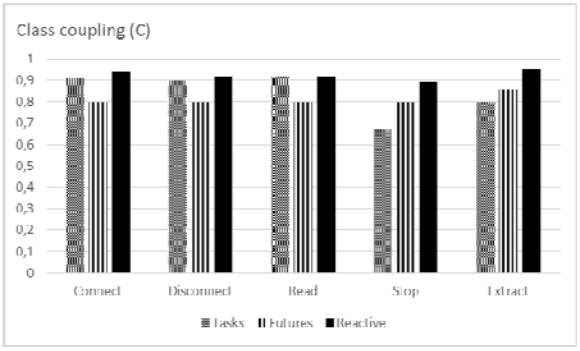

FIGURE 4. CLASS-COUPLING

Figure 5 indicates the run-time performance of the group of methods as implemented in the respective paradigms. In order to reduce probing effects related to time measurement, the data set was chosen to be large, consisting of 100 000 packets.

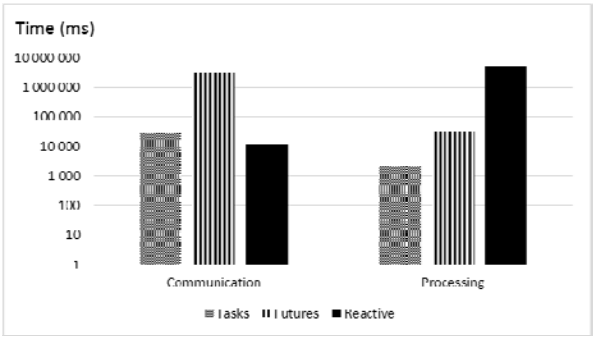

FIGURE 5. PERFORMANCE

IV. EVALUATION

In this section we evaluate the three paradigms in regard to expressiveness, composability and performance.

A. Expressiveness

Regarding expressiveness, both LOC and Cyclomatic complexity have similar results, namely future-based code has a low Cyclomatic complexity number, and requires a limited number LOC. These numbers indicate the expressive power of the future-based paradigm. However this is partly due to the async / await implementation, which generates the required boilerplate code for the continuation. Nevertheless, this pattern is not implementation-specific, as other languages adopt the idea [12].

In contrast, expressiveness of the reactive implementation of the Extract method is significantly low, as indicated by its high Cyclomatic complexity and relative high number of LOC when compared with alternative implementations. However this effect is not reflected when the communication specific methods are considered. In those cases reactive expressiveness is better than the Task-based implementation. This discrepancy can be ascribed to the type of the implemented task. Whereas communication-related methods are event based, the Extract method requires sequential processing of the packets. This reflects in the reactive code, which involves significantly more code to extract data points. However reactive code involving events implicates a small improvement of expressiveness.

B. Composability

In regard to composability, the class-coupling measures indicate a high coupling of the reactive methods. On the other hand the implementation with futures shows a consistently low class-coupling factor. This could be attributed to async /await being a language feature, rather than functionality provided by a library, such as the task and reactive implementations of the respective paradigm. As a library requires the use of a number of classes, it will inevitably enlarge the class-coupling factor. However the language concept will not increase this number, as functionality is provided by the compiler rather than external classes.

C. Efficiency

In overall, the task-based implementation imposes the least overhead, however there are substantial differences in the run-time behavior of the communication related methods and processing data packets.

As the task paradigm is only concerned with the operating mechanisms to schedule tasks, it does not provide any specific mechanisms, such as continuations with futures or event registration and handling with the reactive paradigm. This implies that it is more suited for general purpose problem domains, as it will not impose performance issues in its implementation when dealing with an ill-suited domain. On the other hand, tasks do not provide support in a specific domain.

When compared to tasks, futures will induce overhead. Namely, executing the continuation after a non-blocking wait has been interleaved, requires a number of heap-based operations, which are inevitably an efficiency issue. Nevertheless, due to optimization of the sequential path, this is the path that does not need to wait for the result of the future, the heap operations are avoided which allows to limit the overhead imposed.

Regarding the reactive paradigm, performance of communication related methods indicates the efficiency of the implementation. More specifically, when processing a stream of events, reactive programming has an advantage. However when the reactive paradigm is applied to processing sequential, synchronous data, it is clear that this approach becomes infeasible.

V. Related Work

Van Roy [9] provides an extensive overview of programming paradigms. His taxonomy focusses on state rather than the execution mechanisms. A more in depth review of Reactive programming is given by Amsden [2], and Pembeci and Hager [7]. Their work only focusses on the reactive paradigm and does not provide a comparison to other paradigms.

VI. Future Work

An alternative model of concurrent computation is the Actor model [1]. An Actor is a primitive for a concurrent task, therefore it is potentially alternative for the other task parallel paradigms. Similar to Actors are Models-of-Computation such as SDF, CSDF, PPN, KPN.

Regardless of the task parallel paradigm, the necessity of synchronization between tasks is introduced. Synchronization issues or bugs are hard to detect, due to non-deterministic behavior of concurrent software. The susceptibility towards concurrency issues and difficulty to uncover concurrency bugs were not considered in this study, yet might prove valuable when comparing each paradigm.

VII. Conclusion

In this paper we described three task parallel paradigms, their implications and the evaluation in a case study which concerned data streaming in an embedded system. Choosing a paradigm requires a careful deliberation between expressiveness and composability on the one hand and performance on the other. Moreover, depending on the nature of the application, the preferred choice may shift.

Acknowledgements

We would like to thank Jan Staelens and Brecht Vansteenkiste, both master students in the Department of Industrial Sciences and Technology KHBO, for their participation in the experiments during their final thesis projects.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1985.
[2] E. Amsden. *A survey of functional reactive programming.* Unpublished, 2011.
[3] M. Felleisen. "On the expressive power of programming languages." *ESOP'90*. Springer Berlin Heidelberg, 1990, 134-151.
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[5] D. Leijen, W. Schulte, S. Burckhardt. "The design of a task parallel library." *Acm Sigplan Notices*. 44.10. ACM, 2009.
[6] T. McCabe. "A complexity measure." *Software Engineering, IEEE Transactions on,* 1976, 308-320.
[7] I. Pembeci, G. Hager. "A comparative review of robot programming languages". CIRL lab technical report, Dept. of Computer Science, Johns Hopkins University, 2001.
[8] S. Toub. "Async Performance: Understanding the Costs of Async and Await." *MSDN Magazine-Louisville,* 2011, 34.
[9] P. Van Roy. "Programming paradigms for dummies: What every programmer should know." *New Computational Paradigms for Computer Music*, 2009, 9-47.
[10] S. Verslype, E. Blomme, T. Cool, R. De Craemer, J. Peuteman, J. Vandenbussche. "Realizing high speed LVDS connections in an air-coupled ultrasonic multi channel quality control system." In *Proceedings of the Fourth European Conference on the Use of Modern Information and Communication Technologies*, 2010.
[11] Z. Wan, T. Walid, P. Hudak. "Event-driven FRP". In *Practical Aspects of Declarative Languages*, Springer Berlin Heidelberg, 2002, 155-172.
[12] J. Zaugg, P. Haller. *Simplifying Asynchronous Code with Scala Async*, ScalaDays, 2013.
[13] Curing the asynchronous blues with the Reactive Extensions for .NET. Technical report at Microsoft, 2010.